Computer Physics Communications 263 (2021) 107913

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Efficient GPU implementation of the Particle-in-Cell/Monte-Carlo collisions method for 1D simulation of low-pressure capacitively coupled plasmas[☆]

Zoltan Juhasz [a,*], Ján Ďurian [b], Aranka Derzsi [c], Štefan Matejčík [b], Zoltán Donkó [c], Peter Hartmann [c]

[a] *Department of Electrical Engineering and Information Systems, University of Pannonia, Egyetem u. 10, Veszprem, 8200, Hungary*
[b] *Department of Experimental Physics, Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava, Mlynská Dolina F2, 842 48 Bratislava, Slovak Republic*
[c] *Wigner Research Centre for Physics, Konkoly-Thege M. str. 29-33, Budapest, 1121, Hungary*

ARTICLE INFO

Article history:
Received 2 October 2020
Received in revised form 11 January 2021
Accepted 16 February 2021
Available online 26 February 2021

Keywords:
Collisional plasma simulation
Particle-in-Cell method
GPU

ABSTRACT

In this paper, we describe an efficient, massively parallel GPU implementation strategy for speeding up one-dimensional electrostatic plasma simulations based on the Particle-in-Cell method with Monte-Carlo collisions. Relying on the Roofline performance model, we identify performance-critical points of the program and provide optimised solutions. We use four benchmark cases to verify the correctness of the CUDA and OpenCL implementations and analyse their performance properties on a number of NVIDIA and AMD cards. Plasma parameters computed with both GPU implementations differ not more than 2% from each other and respective literature reference data. Our final implementations reach over 2.6 Tflop/s sustained performance on a single card, and show speed up factors of up to 200 (when using 10 million particles). We demonstrate that GPUs can be very efficiently used for simulating collisional plasmas and argue that their further use will enable performing more accurate simulations in shorter time, increase research productivity and help in advancing the science of plasma simulation.

1. Introduction

Particle-in-cell (PIC) simulations represent a family of numerical methods, where individual particle trajectories are computed in a continuous phase space, while distributions and fields are represented on a numerical mesh. Such a representation of matter is relevant to a large variety of condensed matter and gaseous systems. An early applications of the PIC method was to study the behaviour of hydrodynamic systems [1], subsequently it gained increasing popularity in the field of gas discharge physics after Birdsall and Langdon [2,3] adopted it to electrically charged particle systems including a Monte-Carlo treatment of binary particle collisions (PIC/MCC).

One of the most important fields of applications of the PIC/MCC approach has been the description of capacitively coupled radiofrequency (CCRF) plasmas, which are widely used in high-tech manufacturing, like semiconductor processing, production of photovoltaic devices, treatment of medical implants, like stents, artificial heart valves, etc., see e.g. [4–6]. These plasma sources usually operate in a rarefied gas or gas mixture. The active species, like ions and radicals interact with the samples that are usually connected as electrodes or are placed on the surface of the electrodes. At low pressures, the characteristic free path of the active species, $\lambda = 1/n_0\sigma$, where n_0 is the number density of the background gas and σ is the collision cross section of the active species with the gas atoms/molecules may be comparable or even longer than the dimensions of the plasma source. Under such conditions a correct and accurate description of the particle transport can only be expected from kinetic theory. This, in turn restricts the applicable methods to those that are capable of treating the non-local transport. Particle based methods qualify for this purpose and became therefore a very important approach for the description of low-pressure plasma sources. Their great ability is also to provide (without any *a priori* assumptions) the particle distribution functions which determine the rates of plasmachemical processes (producing the species of interest) and the nature and effectiveness of the reactions taking place at the surfaces to be processed.

The simulation of collisional plasmas (such as those just mentioned) is a computationally very expensive task. When smooth distributions with high spatial and temporal resolution are required, a very large number of particles and collisions needs to be

[☆] The review of this paper was arranged by Prof. David W. Walker.
[*] Corresponding author.
E-mail address: juhasz@virt.uni-pannon.hu (Z. Juhasz).

https://doi.org/10.1016/j.cpc.2021.107913

simulated over long enough time. A good compromise between accuracy and simulation time can be found in cases when the geometrical symmetry of the system enables the reduction of the spatial dimension of the particle phase space from 3D to 2D (in case of cylindrical or slab geometry) or even 1D (in case of plane-parallel systems). The execution time of traditional sequential simulations of one-dimensional systems can vary from several hours to several weeks depending on the simulation parameters [7]. In terms of execution time, sequential simulations of 2D and 3D plasma systems are clearly impractical.

Parallel computing has been used extensively in *collisionless* plasma simulations for decades [8,9]. Interest in efficient parallel implementations increased significantly in the past 10 years as multi-core systems became pervasive [10–13]. Early parallel plasma simulations were typically based on MPI [14] and OpenMP [15] implementations running on large compute clusters or supercomputers, and were characterised by coarse-grain parallelism based on domain decomposition using few hundred to few thousand cores. In special cases, for very large simulations, the number of CPU cores may reach few hundred thousands. As the parallel techniques of collisionless plasma simulation matured, the focus of attention gradually shifted from low-level algorithmic issues to higher-level aspects such as the creation of simulation frameworks and object-oriented solutions [9,16] aiming at simplifying the simulation process for end-users and facilitating code reuse.

Advances in parallel computing technology over the past decade has dramatically changed the hardware architecture landscape. The emergence of general purpose graphics processing units (GPGPUs) gave rise to cards with thousands of compute cores and performance in the range of several Tflop/s (10^{12} operations per second). The majority of supercomputers now employ thousands of GPUs raising the core count to the millions (see e.g. supercomputer Summit with $>$ 140 million GPU cores). These new capabilities resulted in increased interest in massively parallel accelerator based 2D and 3D PIC simulation studies reporting maximum speedup values in the range of 30–50 [17–22].

Despite the success in parallel collisionless plasma simulation, parallel simulation of *collisional* plasma is still a challenge. The difficulty in achieving even modest performance improvements is due to the peculiarities of collisional plasma simulations, such as irregular memory access patterns, the computational cost of parallel random number generation and collision calculation, as well as potentially severe load imbalance. Special memory and particle management schemes are required to achieve increase in performance [23,24].

Several groups examined the use of GPUs for collisional PIC/MCC plasma simulation. Fierro et al. [25,26] report on a 3D PIC/MCC implementation achieving speedups of 13 for the Poisson solver and 60 for the electron mover section of the code. Overall program speedup is not known. Shah et al. [27] developed a 2D simulation for the NVIDIA Kepler architecture and achieved a speed increase of 60 when using 1 million particles. Sohn et al. [28] developed a 2D high-temperature plasma simulation for studying magnetron sputtering achieving speedup of 3–30. Clustre et al. [29] implemented a 2D PIC/MCC GPU code for magnetised plasma simulation achieving speedup in the range of 10–20. Hur et al. [30] also report on 2D magnetised/unmagnetised plasma implementations, with speedups of 80 and 140 for the magnetised/unmagnetised cases, respectively.

Very few teams looked at the GPU implementation of the 1D PIC/MCC case, most likely because a 1D sequential PIC/MCC code computationally is not prohibitive. Nevertheless, shorter runtimes are still important as there are long-running 1D simulations and the decrease in runtime could also allow improving simulation accuracy by increasing particle counts. Mertmann et al. [31] report on a 1D GPU implementation using a novel particle sorting mechanism achieving an overall 15–20x speed increase when using 5 million particles. Hanzlikova [32] studied the efficient implementation of 1D and 2D PIC/MCC algorithms on GPU. The achieved speedup values are not known.

In this paper we show implementation approaches for the fundamental steps of 1D PIC/MCC simulation on GPUs in a massively parallel fashion and examine the performance and scalability of the developed algorithms. We highlight that traditional porting of existing codes is not guaranteed to provide good performance; for the efficient use of GPUs, new programming approaches and new algorithms are required that take architectural characteristics into consideration and provide the level of parallelism needed for these massively parallel systems. We argue that GPU technology is a key enabler for advancing the science of PIC/MCC plasma simulation and its full potential in PIC/MCC simulation has not yet been reached. Chip manufacturing, energy and performance constraints all point to the direction that GPUs will be central elements of our computing systems for many years to come. The unprecedented pace of development in GPU hardware and programming technology continually improves the efficiency of GPU systems, making them suitable for an increasing number of computing problems, while at the same time making their programming simpler. These new algorithms and implementations are also of paramount importance for massively parallel CPU and hybrid peta- and exascale systems as well, since their architectural characteristics and programming challenges are similar to GPUs (e.g. hiding memory access latency and inter-node communication, increasing data locality).

The structure of our paper is as follows. In Section 2 we briefly overview the fundamental concepts of PIC/MCC plasma simulations with the governing equations. Section 3 gives a short introduction to the architecture and programming of GPUs highlighting performance-critical features and to the Roofline performance model. Section 4 provides details of our CUDA and OpenCL 1D PIC/MCC implementations with performance analysis and optimisations. In Section 5, we first show the results of the verification of our implementations by comparing the simulation results with standard benchmark cases, then present performance results of strong and weak scaling cases as well as overall speedup values and analysis of system performance in function of particle counts. The paper ends with the conclusions.

2. Basics of the PIC/MCC simulation of CCRF plasmas

Typical technological CCRF plasma sources are excited by single- or multifrequency waveforms at frequencies between 1 MHz and 100 MHz, with voltage amplitudes of several hundred Volts. At "low-pressure" conditions they operate between $\sim$ 1 Pa and few hundred Pa. The typical electron and ion density in these plasmas is in the order of $n_e \approx n_i \sim 10^8 - 10^{11}$ cm^{-3}. Considering plasma volumes of $\sim 10^2 - 10^3$ cm^3 the number of electrons/ions may be in the order of up to $N \sim 10^{14}$. These particles move under the influence of their mutual interactions and the external electric field that accelerates the charged species thereby establishing the plasma. Accounting fully for these interactions, especially the pairwise interactions, is impossible, therefore the following two simplifications are conventionally introduced: (i) the direct particle–particle pair interaction is neglected, the particles move in an electric field that results from the external field and from the field generated by the presence of the charged particles, (ii) instead of each and every particle, "superparticles" are traced that represent a large number of real particles. These simplifications form the basis of the "Particle-in-Cell" (PIC) simulation method. In PIC simulations plasma characteristics (like charged particle densities, electric field strength) are computed at points of a spatial grid, at discrete times.

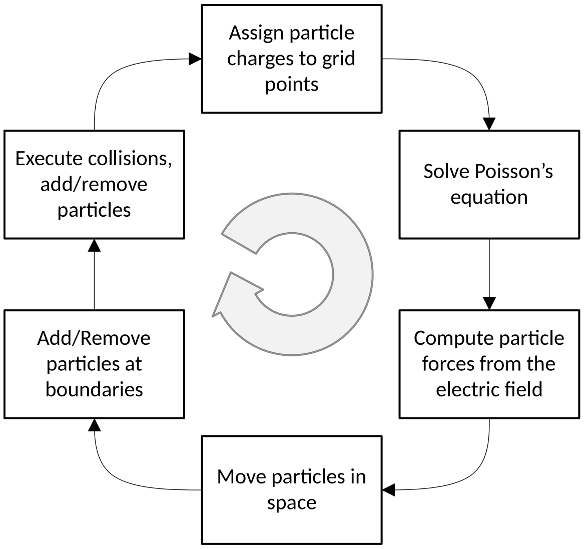
Assign particle charges to grid points
Solve Poisson's equation
Compute particle forces from the electric field
Move particles in space
Add/Remove particles at boundaries
Execute collisions, add/remove particles

Fig. 1. The simulation cycle of a PIC/MCC bounded collisional plasma that has to be carried out several thousand times within a radiofrequency period.

As the plasmas of our interest are collisional, the PIC method (originally developed for collisionless plasmas) has to be complemented with the description of collision processes (as already mentioned in Section 1). For this purpose a stochastic treatment, the "Monte Carlo Collisions" (MCC) approach is usually adopted [3]. In this approach, colliding particles are selected in each time step randomly, based (strictly) on the probabilities of such events defined by the respective cross sections, the velocities of the particles, the background gas density and the length of the time step (see later).

The PIC/MCC method is thus a combination of two approaches. It is capable of describing electrodynamics effects, but in most cases the electrostatic approximation provides sufficient accuracy. The method is readily applicable to 1, 2, or 3 dimensions, however, the computational load increases immensely for the higher (2 or 3) dimensions. In the following we assume a one-dimensional geometry: a source in which the plasma is established between two plane parallel electrodes, which are placed at a distance L from each other. The "lateral" extent of the electrodes is assumed to be infinite, the plasma characteristics are therefore functions of the x coordinate only. The velocity of the charged particles (electrons and ions are considered), is however computed in the three-dimensional velocity space upon collisions. Because of the cylindrical symmetry of the system, however, the particles carry only the values of their axial and radial velocities, v_x and v_r, respectively, between their collisions with the atoms of the background gas. The superparticle weight W (that expresses how many real particles are represented by a superparticle) is the same for electrons and ions.

The electrostatic PIC/MCC scheme considered here consists of the following "elementary" steps (see Fig. 1), which are repeated in each time step:

- computation of the density of the particles and the total charge density at grid points,
- calculation of the potential distribution from the Poisson equation that contains the potentials of the electrodes as boundary conditions,
- interpolation of the computed electric field to the position of the particles,
- moving the particles as dictated by the equations of motion,
- identification of the particles that reach the surfaces, accounting for the interaction with the surfaces and removing them from the simulation,
- checking of the collision probabilities of the particles and executing the collision processes.

Here, we adopt an equidistant grid that has a resolution of $\Delta x = L/(M-1)$, where M is the number of the grid points having coordinates x_p with $p = 0, 1, 2, \ldots M-1$. The densities of charged particles at the points of this grid are computed with linear interpolations according to the particles' positions as described below. The procedure applies both to electrons and ions. If the jth particle is located within the pth grid cell ($p = [x_j/\Delta x]$, where x_j is the position of the jth particle (equal either to $x_{\mathrm{e},j}$ or $x_{\mathrm{i},j}$ for an electron/ion, respectively) and $[\cdot]$ denotes the integer part), the density increments assigned to the two grid points that surround particle j are

$$\delta n_p = \big((p+1)\Delta x - x_j\big)\frac{W}{A\Delta x^2}, \tag{1}$$

$$\delta n_{p+1} = (x_j - p\Delta x)\frac{W}{A\Delta x^2}, \tag{2}$$

where A is the fictive electrode area (that is required for normalisation purposes only, as the "real" area of the electrodes is infinite). The linear interpolation scheme provides a good balance between performance and accuracy, hence why it has become the standard way to compute the particle densities at the grid points. In specific cases higher-order interpolation schemes may be used [2].

Once the particle densities at the grid points are known, the charge density at the grid points can be computed as

$$\rho_p = e(n_{\mathrm{i},p} - n_{\mathrm{e},p}), \tag{3}$$

where e is the elementary charge, $n_{\mathrm{i},p}$ and $n_{\mathrm{e},p}$ are, respectively the ion and electron densities at grid point p. The potential distribution is obtained from the Poisson equation:

$$\nabla^2\phi = -\frac{\rho}{\varepsilon_0}. \tag{4}$$

This equation can be rewritten in the following finite difference form for one-dimensional problems:

$$\frac{-\phi_{p-1} + 2\phi_p - \phi_{p+1}}{\Delta x^2} = \frac{\rho_p}{\varepsilon_0}, \tag{5}$$

known as the Discrete Poisson Equation. Solving this equation is described in further detail in Section 4.6. Differentiating the potential distribution (Eq. (6)) we obtain the electric field at each grid point. Boundary grid points need to be treated specially as given by Eqs. (7) and (8):

$$E_p = \frac{\phi_{p-1} - \phi_{p+1}}{2\Delta x}, \tag{6}$$

$$E_0 = \frac{\phi_0 - \phi_1}{\Delta x} - \rho_0\frac{\Delta x}{2\varepsilon_0}, \tag{7}$$

$$E_{M-1} = \frac{\phi_{M-2} - \phi_{M-1}}{\Delta x} + \rho_{M-1}\frac{\Delta x}{2\varepsilon_0}. \tag{8}$$

Knowing the electric field values at the grid points allows computation of the field at the position of particle j located at x_j, as

$$E(x_j) = \frac{(p+1)\Delta x - x_j}{\Delta x}E(x_p) + \frac{x_j - p\Delta x}{\Delta x}E(x_{p+1}). \tag{9}$$

The force being exerted on the jth particle located at x_j are given by

$$F_j = q_j E(x_j), \tag{10}$$

Table 1
Physical and numerical parameters of the benchmark cases.

		Case			
		1	2	3	4
Physical parameters:					
Electrode separation	L [cm]	6.7	6.7	6.7	6.7
Gas pressure	p [mTorr]	30	100	300	1000
Gas temperature	T [K]	300	300	300	300
Voltage amplitude	V [V]	450	200	150	120
Frequency	f [MHz]	13.56	13.56	13.56	13.56
Numerical parameters:					
Cell size	Δx	$L/128$	$L/256$	$L/512$	$L/512$
Time step size	Δt	$f^{-1}/400$	$f^{-1}/800$	$f^{-1}/1600$	$f^{-1}/3200$
Total steps to compute	N_{S}	512 000	4 096 000	8 192 000	49 152 000
Data acquisition steps	N_{A}	12 800	25 600	51 200	102 400
Particle weight factor	W	26 172	52 344	52 344	78 515
Converged computed parameters:					
Number of electrons	N_{e}	12 300	57 000	138 700	161 600
Number of ions	N_{i}	19 300	60 200	142 300	164 500
Peak ion density	$\hat{n}_{\mathrm{i}}$ [10^{15} m^{-3}]	0.140	0.828	1.81	2.57

resulting in an acceleration

$$a_j{}^t = \frac{q_j}{m_j} E(x_j), \tag{11}$$

which makes it possible to solve the equations of motion using, e.g., the popular leap-frog scheme, where the particle positions and accelerations are defined at integer time steps, while the velocities are defined at half integer time steps:

$$v_j{}^{t+1/2} = v_j{}^{t-1/2} + a_j{}^t \Delta t, \tag{12}$$

$$x_j{}^{t+1} = x_j{}^t + v_j{}^{t+1/2} \Delta t. \tag{13}$$

After updating the positions of the particles a check must be performed to identify those particles that have left the computational domain. These particles have to be removed from the ensemble.

At every time step, the probability P_j of the jth particle undergoing a collision is given by [3]

$$P_j = 1 - \exp\left[n_{\mathrm{g}}(x)\sigma_{\mathrm{T}}(\varepsilon)v_j\Delta t\right] \tag{14}$$

where $n_g(x)$ is the background gas density, $\sigma_{\mathrm{T}}(\varepsilon)$ is the particle's energy-dependent total cross-section, v_j is the jth particle's velocity and Δt is the simulation time step. Based on the comparison of P_j with a random number (having a uniform distribution over the $[0, 1)$ interval) decision is made upon the occurrence of the collision. The total cross-section is given by the sum of cross-sections of all collisions the specific particle can participate in:

$$\sigma_{\mathrm{T}}(\varepsilon) = \sum_{s=0}^{k} \sigma_s(\varepsilon), \tag{15}$$

where k is the number of reaction channels for the species of the specific particle.

There is a finite probability that a particle undergoes more than one collision within a single time step, resulting in missed collisions. For this reason the time step of the simulation must be chosen to be sufficiently small, as given by [33]:

$$\nu_{\max} \Delta t \ll 1 \tag{16}$$

where $\nu_{\max}$ is the maximum of the energy dependent collision frequency $\nu(\varepsilon) = n_{\mathrm{g}}\sigma_{\mathrm{T}}(\varepsilon)v$. Choosing a time step small enough for $P_j \approx 0.1$ results in a sufficiently small number of missed collisions.

The type of the collision that occurs is chosen randomly, on the basis of the values of the cross sections of the individual possible reactions at the energy of the colliding particle. The deflection of the particles in the collisions is also handled in a stochastic manner, see e.g. [34].

Details of the implementations of the steps described above will be discussed in parts of Section 4.

The steps outlined above have to be repeated typically several thousand times per RF cycle. For sufficient accuracy the number of superparticles has to be in the order of $\sim 10^5 - 10^6$ in 1D simulations. The spatial grid normally consist of $100 - 1000$ points. These values are dictated by the stability and accuracy requirements of the method (which are not discussed here but can be found in a number of works, e.g. [33]). Convergence to periodic steady-state condition is typically reached after thousands of RF cycles. Due to these computational requirements, in the past, such simulations were only possible on mainframe computers. Today, *sequential* 1D simulations can be carried out on PC-class computers or workstations, with typical execution times between several hours and several weeks (depending mainly on the type and the pressure of the gas).

Ideally, the results of a PIC/MCC simulation should not depend on the number of superparticles used. In reality, however, as it was discussed by Turner [35] low particle numbers give incorrect results due to numerical diffusion in velocity space. The dependence of the simulation results on the number of computational particles was also analysed in [36]. The number of superparticles in our work primarily follows those used in [7], which is appropriate for a benchmarking activity, but we also executed performance measurements for higher numbers of superparticles. It should, however, be kept in mind that for results with improved absolute accuracy the higher particle numbers should be adopted, at least in the range of few times 10^5 per species.

2.1. Benchmarks

To perform the validation of our GPU implementations we adopt the four benchmark cases that have been introduced in [7] and compare the computed charged particle densities, being some of the most fundamental and very sensitive plasma parameters, with the reference data of [7]. The parameters of the plasma benchmark cases 1–4 are listed in Table 1.

The discharges are assumed to operate in plane-parallel geometry with one grounded and one powered electrode facing each other, therefore a spatially one-dimensional simulation can be applied. The discharge gap is filled with helium gas at densities and temperature (300 K) that are fixed for each benchmark case. A harmonic voltage waveform is applied to the powered electrode at a frequency of 13.56 MHz. The voltage amplitudes are given for each of the four benchmark cases. Charged particles

Table 2
The collision types used in our plasma model.

Reaction	Type	Threshold energy (eV)
e + He → e + He	Elastic	–
e + He → e + He*	Excitation (triplet)	19.82
e + He → e + He*	Excitation (singlet)	20.61
e + He → e + e + He^+	Ionisation	24.59
He^+ + He → He^+ + He	Elastic (isotropic part)	–
He^+ + He → He + He^+	Elastic (backward part)	–

Table 3
Comparison of CUDA and OpenCL terminology.

CUDA	OpenCL
GPU (device)	Device
Multiprocessor	Compute Unit
Scalar core	Processing Element
Kernel	Kernel
Global memory	Global memory
Shared memory	Local memory
Local memory	Private memory
Grid	Computation domain
Thread block	Work-group
Warp	Wavefront
Thread	Work-item

reaching the electrodes are absorbed, and no secondary electrons are emitted. The simulated plasma species are restricted to electrons and singly charged monomer He^+ ions. Collisional processes are limited to interactions between these species and neutral helium atoms. For electron-neutral collisions, the cross section compilation known as "Biagi 7.1" is used [37]. This set includes elastic scattering, excitations and ionisation. Isotropic scattering in the centre of mass frame is assumed for all processes. For the ions only elastic collisions are taken into account, with an isotropic and a backward scattering channel, as proposed in [38]. The cross sections for the ions are adopted from [39]. The list of the processes is compiled in Table 2.

3. GPU programming and architecture

Modern GPUs can be programmed in a variety of ways. OpenACC [40] and OpenMP 4.x [15,41] provide compiler-based directives that allow a simple and iterative parallelisation of existing sequential programs, particularly if they are based on algorithms with tightly nested loops. OpenMP is platform-neutral while OpenACC has a certain bias towards NVIDIA GPUs. Developers requiring tighter control of hardware resources and a more programmatic approach, can use GPU programming languages such as OpenCL [42,43] or CUDA [44,45]. These are more suited to complex algorithms and advanced performance optimisation but at the expense of increased code complexity and programming effort. OpenCL is a hardware agnostic standard supported by various vendors, utilising a portable C-based language to generate code executable on CPUs (x86/x64 and ARM architectures alike), and various accelerators including NVIDIA, AMD and Intel GPUs, and FPGAs. CUDA has a C/C++ and Fortran programming interface and can only be used on NVIDIA GPUs. To maximise programming flexibility and performance, we chose a programming language approach over directives and used both CUDA-C and OpenCL for our GPU implementations. Since the CUDA and OpenCL terminology are slightly different, in the rest of the paper we will use CUDA terminology; the matching OpenCL terms can be looked up in Table 3.

Common in both CUDA and OpenCL is the concept of the *kernel*, a GPU function called from the host program and executed in parallel by multiple GPU threads (a.k.a. the SIMT /Single Instruction Multiple Thread/ model). Kernel execution is an asynchronous operation; the CPU and GPU can execute instructions simultaneously. Synchronisation constructs are available if the GPU and CPU instructions must follow a prescribed execution order.

Every kernel call must contain two parts, (i) the list of function arguments found also in standard C functions, and (ii) the execution parameters which define the degree of parallelism. These execution parameters are determined by an object called *grid* (not to be confused with a computational grid used in the PIC algorithm), which defines an index space for the individual threads to use. In theory, these grids can be N-dimensional, however, modern GPUs only support grids of up to three dimensions. All launched threads are assigned a unique N-dimensional index from this grid, and executed on the GPU in groups called thread blocks.

From the architectural point of view, each GPU device consists of several Graphics Processing Clusters (GPCs) that contain a number of Streaming Multiprocessors (SMs). Each SM contains several compute cores; in modern GPUs, there are separate floating-point (FP32 and FP64), integer (INT32), special function unit (SPU) and tensor cores. These cores execute the actual instructions of a launched kernel. The number of cores is computed as the product of the number of GPCs, the number of SMs per GPC and the number of cores per SM. For instance, the number of cores in the NVIDIA V100 accelerator card is given as 6 GPCs x 10 SMs/GPC x 64 FP32/SM = 5120. If we multiply the number of cores by the clock frequency of the GPU and a factor of 2 (assuming the execution of fused multiply-add /FMA/ instructions only that count as 2 operations per cycle), we obtain the approximate peak performance value (e.g. V100 card: 5120 cores × 2 instructions/cycle × 1.4 GHz ≈ 14 TFlop/s).

Each SM has a large register file (256 kB) providing 64k registers or up to 255 registers per thread, a fast on-chip shared memory (48 or 64 kB) and an L1 cache. The shared memory is used for fast data sharing among threads running on the same SM. Instructions are scheduled by up to four thread schedulers. Since global memory operations take in the order of 600–900 clock cycles to complete, a large number of threads is required to hide data latency. The schedulers choose threads for execution that are not stalled on data or synchronisation operations. In addition, advanced GPUs support concurrent kernel execution, as well as a number of instruction streams that help developers to maximise GPU utilisation.

3.1. Hardware environment

A wide range of GPU devices are available on the market ranging from gaming cards to datacentre/supercomputer accelerators. In this paper we have used a number of GPUs representing different product classes to demonstrate their effectiveness in PIC/MCC plasma simulation and to compare their relative performance. Table 4 contains the details of the selected set of GPU cards that we used during the implementation and performance evaluation. For each card, the table includes the number of FP32 cores, the peak computational performance, the GPU memory bandwidth values, and the HW instruction-to-byte ratio – computed as *peak performance* (Gflop/s) / *peak bandwidth* (GB/s) – that forms the basis of the roofline performance analysis method that we use later in the paper.

The roofline performance model [46,47] provides a simple but effective tool for estimating the attainable performance and exploring the expected behaviour of parallel algorithms. If the Arithmetic Intensity (number of executed FP instructions/number of bytes read and stored) of a kernel is greater than the HW

Table 4
Performance characteristics of the selected GPU cards.

GPU card	Cores	Instruction throughput (GFlop/s)	Bandwidth (GB/s)	HW instruction-to-byte ratio
NVIDIA GTX 1080 Ti	3584	11 339	484	21.91
NVIDIA GTX 1660 S	1408	5027	336	25.64
NVIDIA RTX 2080 Ti	4352	13 447	616	19.91
NVIDIA Tesla P100	3584	9340	549	14.70
NVIDIA Tesla V100	5120	14 899	900	16.55
AMD Radeon Pro WX9100	4096	12 290	483.8	25.40
AMD Radeon VII	3840	13 440	1024	13.125

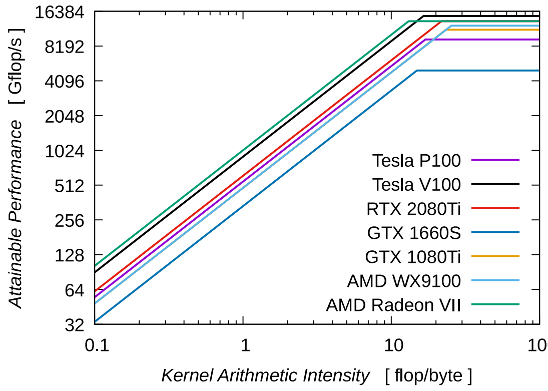

Fig. 2. Roofline performance model of the selected GPU cards. Kernels with arithmetic intensity below the HW instruction-to-byte ratio are memory-bound; the attained performance is limited by the available memory bandwidth.

instruction-to-byte ratio, the kernel is compute-bound and can potentially reach near peak performance, otherwise it is memory-bound and the performance is determined by the achieved memory bandwidth. More formally, the performance is given as

$$\begin{aligned}&\mathit{AttainablePerformance}\,[\mathrm{GFlop/sec}]\\&\quad=\min(\mathit{PeakFloatingPointPerformance},\\&\qquad \mathit{PeakMemoryBandwidth}\times\mathit{ArithmeticIntensity}),\end{aligned}\tag{17}$$

where *ArithmeticIntensity* (*AI*) is the ratio of the executed floating point operations and the bytes transferred to/from memory in the given GPU kernel,

$$\mathit{ArithmeticIntensity}=\frac{\mathit{FloatingPointOperations}}{\mathit{Bytes}_{\mathrm{read}}}+\mathit{Bytes}_{\mathrm{written}}.\tag{18}$$

Some of the above parameters can be obtained from GPU specifications while others are estimated from the algorithm. When the attainable performance is plotted in function of the arithmetic intensity, a curve resembling a roof line emerges. The intersection of the two lines is at the arithmetic intensity threshold (the HW instruction-to-byte listed in Table 4) that a code must achieve to become compute-bound. Fig. 2 shows the roofline curves of the six selected GPU devices used in this paper. Note that an Arithmetic Intensity in the range of 13–26 is required to achieve maximum compute performance on these GPUs.

4. GPU implementation

In this section we discuss the details of the parallel GPU implementation of the PIC/MCC simulation core. To keep the description short and easy to follow, we will not include every detail but concentrate only on the most crucial issues. Emphasis is placed on performance-critical design and implementation decisions.

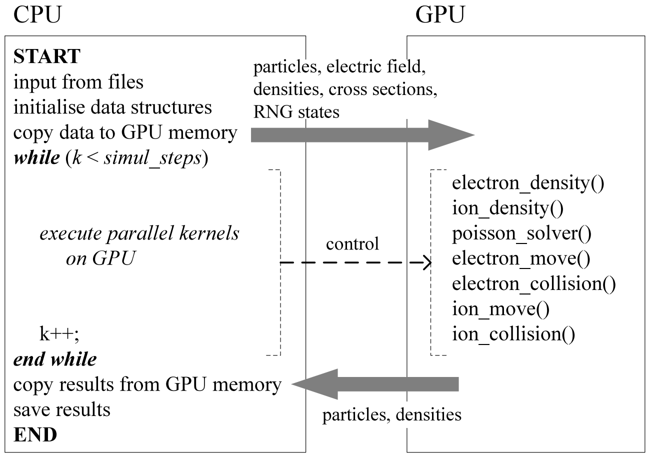

Fig. 3. High-level structure of the parallel PIC/MCC simulation program. The host system is responsible for initialising the simulation and saving results. The actual simulation is executed exclusively on the GPU by a sequence of GPU kernel calls. Note that the final version uses restructured kernels as described in Section 4.7.

4.1. Design principles

The primary goal of our work is to maximally harness the performance of the GPUs in PIC/MCC simulations. The two guiding principles in achieving this are (i) minimising execution overheads and (ii) maximising available parallelism.

As depicted in Fig. 3, our implementation uses a GPU-only simulation loop. The CPU code is used only to call the execution of the parallel GPU kernels (electron_density(), ion_density(), and so on). Since PIC simulations execute a very large number of simulation loop iterations, it is important to remove any unnecessary CPU–GPU communication inside the loop, since the PCI-e bus is a performance critical point of GPU systems. Several authors implemented hybrid systems, executing parts of the loop (e.g. the Poisson solver) on the CPU [21,48] to keep the implementation simpler. Even if the execution time of a code section might be faster on a CPU, taking all costs (CPU execution time, data transfer and synchronisation overheads) into account, the end result may not be ideal; performance suffers. We focused on implementing each step of the simulation loop on the GPU even at the cost of higher code complexity and increased programming effort.

Maximising the level of parallelism required an architecture-oriented perspective during the code design. Traditional CPU-based parallel PIC plasma simulations are based on partitioning the physical domain according to the number of cores, where each thread loops over the particles contained in a given partition. In the 1D plasma simulation case, this coarse-grain parallelism offers speedups up to the order of 10^2. GPU chips, however, contain thousands of cores, generating a mismatch between the partitioning scheme and the underlying architecture. In addition, global memory operations have large latency, $> 600 - 800$ clock cycles. To hide the cost of memory access, GPU programs require

Table 5
Dictionary of the notations of the physical quantities, which appear both in the theoretical part (Section 2) and in the parts of the code in this section.

Quantity	Notation	Variable in code
Electric field	$E(x)$	e_field[ ]
Number of electrons	N_e	N_e
Position of an electron	$x_{e,j}$	x_e[ ]
Axial electron velocity	$v_{x,e}(x)$	vx_e[ ]
Radial electron velocity	$v_{r,e}(x)$	vr_e[ ]
Electron density	$n_e(x)$	e_density[ ]
Number of positive ions	N_i	N_i
Position of an ion	$x_{i,j}$	x_i[ ]
Axial ion velocity	$v_{x,i}(x)$	vx_i[ ]
Radial ion velocity	$v_{r,i}(x)$	vr_i[ ]
Positive ion density	$n_i(x)$	i_density[ ]
Number of grid points	M	M
Superparticle weight	W	weight
Simulation time step	Δt	dt
Division of spatial grid	Δx	deltax
Electron impact total cross section	$\sigma_e(\varepsilon)$	sigma_e[ ]
Ion impact total cross section	$\sigma_i(\varepsilon)$	sigma_i[ ]

significantly more threads than cores for efficient execution. As a rule of thumb, $10^5 - 10^6$ concurrent threads are recommended for chips with few thousand cores. Consequently, instead of partitioning the physical domain, we map each particle onto an individual thread.

In the rest of this section we describe the key features of our parallel GPU implementation and discuss them from a performance optimisation perspective. The order of treatment is not the order of execution in the simulation loop; we first discuss those kernels that show performance problems but at the same time provide optimisation opportunities. Kernels that perform well or offer limited optimisations are described afterwards. To help readers connecting the implementation details to the theory and equations outlined in Section 2, in Table 5 we created a dictionary mapping the notation of physical quantities to the corresponding program variables.

4.2. Data structures and memory layout

The state of each type of superparticle includes its position and velocity components. In a Cartesian system, the state can be described by the tuple $\langle x, v_x, v_y, v_z \rangle$. In a cylindrical coordinate system, this simplifies to $\langle x, v_x, v_r \rangle$. The plasma characteristics: the density of the charged species, the potential as well as the electric field are computed at the grid points.

There are several alternative schemes for storing particle state values. Software engineering best practices would suggest using a C structure (or a C++ class) encapsulating position, velocities and additional particle information which then can be stored in an array or list data structure (Array of Structures, AoS). Another alternative is to use special GPU data types, such as `float4` having `x`, `y`, `z`, `w` elements to represent a tuple and creating arrays of these elements. The third option is to use separate N-element float arrays for individual elements of the tuple.

GPU architecture is designed in a way that memory is accessed the most efficient way if consecutive threads in a thread warp access consecutive memory addresses. In other words, with this ideal, aligned and coalesced addressing pattern, a 32-thread warp can read 128 (i.e. 32 × 4) bytes either as one 128-byte or four 32-byte transactions, depending on the given GPU architecture generation. This access pattern can only be guaranteed with separate arrays for position and velocity components. Using plain C arrays contradicts common high-level programming practices that promote encapsulation but demonstrates that high-performance parallel code often requires deviation from the software engineering methods of the sequential world. Other memory layouts would introduce various memory bank conflicts that drastically increase the required number of memory transactions. Since memory and computational performance is maximised if float data types are used instead of doubles, our program defaults to floats except for the density calculation and Poisson solver code sections.

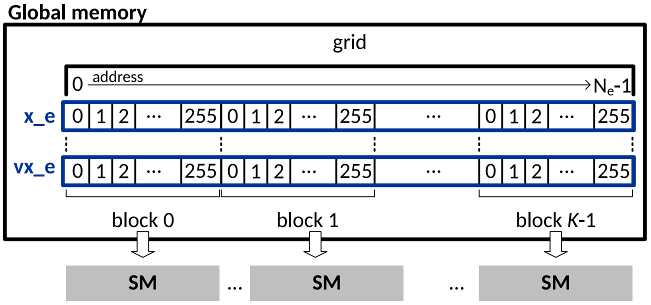

Fig. 4. Data structures and their layout in GPU memory (only the electron position (x_e) and axial velocity (vx_e) arrays are shown). K blocks of 256 threads form a thread grid that are used to process all data elements. Within the blocks each number represents a thread index. Blocks are mapped to available streaming multiprocessors by the thread schedulers of the GPU. Memory access is coalesced as threads in each warp read contiguous memory addresses.

Fig. 4 shows the data structures used in our implementations along with illustrations how the GPU threads are mapped in units of thread blocks onto the array elements. We only show the position and velocity arrays for electrons holding `N_e` electrons. Storing particle data in this manner results in ideal, aligned and coalesced data load/store access pattern.

Besides access patterns, memory access efficiency is also dependent on the size of the transferred data. As shown in Fig. 5, the effective memory bandwidth can be very different from the peak bandwidth values found in hardware specifications. The plots show the global memory bandwidth derived from kernel load/store operation performance in function of particle numbers. Depending on the card chosen, a minimum of 250k to 1M particles are required for reaching maximum bandwidth. It is worth noting that most cards have a relatively narrow 'sweet spot' range where the bandwidth is higher than the peak value. This is especially pronounced in for RTX 2080Ti card where the range is much wider (300–800k particle). We suspect this is due to memory clock boost effects and the caching mechanism. Kernels with particle counts below 100k show significantly reduced memory bandwidth, reducing the performance of memory-bound kernels even further. Fortunately, the drive for increasing the number of particles in simulations meets GPU architectural characteristics and is expected to result in increased performance.

4.3. Particle mover

In a PIC/MCC simulation, particles interact via an electric field E formed by the superposition of an external electric field applied at the electrodes and the electric field created by the charged particles themselves. These interactions are facilitated by a computational grid in which we store information about the value of the electric field. Moreover, in every point of this grid we also store information about the electric potential ϕ, as well as the particle densities.

The particle mover is implemented by the GPU kernel code section below. Each GPU thread of the kernel implementing this simulation step is responsible for one particle, therefore must read one element of the input arrays `x_e[tid]`, `vx_e[tid]`, `vr_e[tid]` for electrons, and `x_i[tid]`, `vx_i[tid]`|, `vr_i[tid]` for ions, where `tid` is the thread index. In addition,

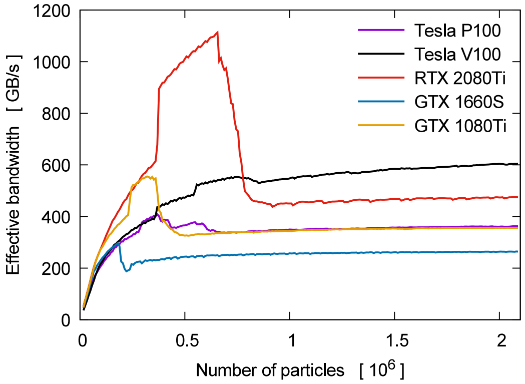

Fig. 5. Effective GPU global memory bandwidth (GB/s) as a function of particles to load/store.

the position `x_e[tid]` or `x_i[tid]` is used to compute the index `p` of the grid cell within which the particle is located. Once the grid cell is located, the interpolated electric field value `e_x` is computed using `e_field[p]` and `e_field[p+1]`, from which the new velocity and position is computed by a leapfrog integration scheme. The lines computing the new velocity `vxei` and position `xei` values (Eqs. (12) and (13) use pre-computed constants `s1` = $q_j \Delta t / m_j$ and `dt_e` = Δt).

```
// compute thread (particle) index
int tid = blockIdx.x * blockDim.x + threadIdx.x;
// load position and velocity values into register variables
xei = x_e[tid]; vxei = vx_e[tid]; vrei = vr_e[tid];
// obtain grid cell index in which the particle is located
int p = xei / deltax;
// calculate ...
float e_x = (((p+1)*deltax-xei) * efield[p] +
         (xei-p*deltax) * efield[p+1])/deltax;
// compute new velocity
vxei -= s1*e_x;
// compute new position
xei += vxei*dt;
check_boundary();
// store new values in global memory
x_e[tid] = xei; vx_e[tid] = vxei; vr_e[tid] = vrei;
```

The next step is to handle particles at the boundary of the system. When a particle leaves the system, a helper index array element (`e_index[i]` or `i_index[i]` for electrons and ions, respectively) is set to 1 to indicate to downstream stages of the calculation that this particle should be ignored in the simulation. In our OpenCL implementation, instead of defining extra arrays, we used the fourth component of the particle velocity vector (`float4` datatype) to mark particles to be ignored. The kernel completes by writing out the new position and velocity values and the random number state into global memory. Our kernel is prepared for treating particles reflected back from the boundary plates, but we ignore this case for brevity.

Now, we turn our attention to the performance of this kernel. The code ideally issues 7 load (`x`, `vx`, `vr`, one index element, random number state and two field values) and 4 store (`x`, `vx`, `vr`, random number state) requests (1 request implies 4 32-bit global memory transactions) and a number of floating-point operations (non floating-point operations are treated as overhead). Instead of hand-counting operations, profiler tools such as NVIDIA `nvprof` and Nsight Compute can be used to obtain the important performance metrics. In the rest of the paper, we will use the following command and a P100 card to obtain metrics.

```
>nvprof --metrics flop_count_sp
     --metrics dram_read_transactions
     --metrics dram_write_transactions
     ./executable input_parameters
```

From these, *ArithmeticIntensity* is computed as *AI* = *flop_count_sp*/[(*dram_read_transactions* + *dram_write_transactions*) ∗ 32] and gives 1.18 (electrons) and 1.16 (ions). The achieved Gflop/s rates (*flop_count_sp*/*kernel_time*) on a P100 card are 150.3 (electrons) and 98 Gflop/s (ions). Fig. 2 shows that the arithmetic intensity value of 1.16 should result in approximately 500 Gflop/s performance on the P100. The low performance of this kernel is the result of the irregular memory indexing by `p` and `p+1` into the array `efield`, which causes unpredictable memory bank conflicts decreasing memory transfer performance. Nsight Compute reveals that the actual average number of memory transactions for this kernel are 35.57 (electrons) and 42.85 (ions) for loads and 12.03 (electrons) and 12 (ions) for stores. The store transactions are less than the required 16 because threads whose particles leave the system will not save output values.

Since the 1.16 arithmetic intensity is well below the optimal arithmetic intensity range of 13–26 needed for a compute-bound kernel (see Table 4 for details) we need to find ways to improve the memory data transfer performance. By utilising the on-chip shared memory, it is possible to reduce global memory bank conflicts. We modify the kernel that each thread block will read the efield array into a private shared-memory copy, then this fast shared field array will be used in the interpolation.

```
extern __shared__ float s_efield[];
int k = threadIdx.x;
while (k < n) {
  s_efield[k] = efield[k];
  k += blockDim.x;
}
__syncthreads();
// use s_efield instead of the global efield
// from this point on
float e_x = ...
```

The benefits are that the global field array will be read in an optimal coalesced way and – while bank conflicts will remain in reading the private arrays – the much lower latency of the shared memory will improve data access performance. As a result, the number of load and store request are reduced to 5.6 and 4 and the number of transactions to 19.1 load and 12 store transactions for both electrons and ions. The arithmetic intensity remained 1.16, but the achieved performance of the kernel increased to 495.5 (electrons) and 486.6 (ions) Gflop/s, which is very close to the approx. 512 Gflop/s limit.

4.4. Collisions

The collision calculation kernel issues 8 load and 4 store requests for electrons, 7 load and 4 store requests for ions, resulting in 65 and 28 as well as 51 and 28 load/store transactions for the two types of particles. These are about twice as high as the required number of transactions at coalesced memory access. The cause of this inefficiency lies in loading the σ_e and σ_i cross-section values from memory. The cross sections are stored as interpolated table values that are looked up during the computation by indirect indexing that causes bank conflicts. In addition, since not all particles collide, thread predication due to conditional code branching also has a performance decreasing effect.

The arithmetic intensity of the electron collision kernel is 0.65, the achieved performance is 256.5 Gflop/s. The ion collision code performs better, due to its higher intensity (*AI* = 2.92) reaching 1205.6 Gflop/s performance. While both kernels show similar memory access behaviour, the higher arithmetic load in the ion collision calculation compensates memory inefficiencies to a certain degree.

4.4.1. Random number generation

The quality and speed of random number generation in parallel Monte Carlo methods is of prime importance. In our first implementations we used vendor and community developed generators (CUDA: cuRand, OpenCL: clRNG libraries) to create uniformly distributed random sequences for each particle. These implementations however seemed to be computationally too expensive. In the final version, we used $N = N_e + N_i$ independent sequences by initialising N generators based on the standard `ran` implementation [49] with different seeds. The statistical properties of this method were tested and found correct for our purposes. The key benefit is the approximately 3× speed gain over the library versions. For the particle number range 64k–1M, the cuRand XORWOW execution times are 35–3158 µs, whereas in our implementation are 13.5–974 µs.

4.5. Density calculations

In order to be able to calculate the potential and the electric field at the grid points the charge density corresponding to the electrons and ions has to be determined. In our scheme we distribute the density of each charged particle to two grid points between which the particle resides. The density is weighted to these two points according to the distances between the particle and the grid points. Taking one electron as example, this "charge assignment" procedure could be executed by the following lines of the code. The variable `c_de_factor` holds the pre-computed value of $W/A\Delta x^2$ stored in GPU constant memory for efficient broadcasting to all threads.

```
 // compute thread (particle) index
int tid = blockIdx.x * blockDim.x + threadIdx.x;
// load particle position
xei = x_e[tid];
// find grid cell index
p = (int)(xei/deltax);
// calculate charge for grid points p and p+1
e_density[p] += (p+1)*deltax-xei)*c_de_factor;
e_density[p+1] += (xei-p*deltax)*c_de_factor;
```

The update instructions in the last two lines present a challenging situation. Since all threads execute concurrently, appropriate locking mechanisms must be used to protect memory writes from race conditions. GPUs provide various atomic operations to handle these cases. The simplest solution is to replace the $+=$ operators with atomicAdd function calls. These execute the global memory read–modify–write cycle instruction sequence efficiently with built-in hardware support.

```
atomicAdd(&(e_density[p]), ((p+1)*deltax-xei)*c_de_factor);
atomicAdd(&(e_density[p+1]), (xei-p*deltax)*c_de_factor);
```

The electron and ion density kernels have reasonably high arithmetic intensities (2.69 and 3.98, respectively) but the achieved performance is rather low: 42.7 and 37.8 GFlop/s. This is the result of executing a large number of global memory atomic transactions at conflicting addresses.

It is possible to improve this code by executing a parallel reduction operation. Each thread block (i.e. SM) will create a private density array in its on-chip shared memory as illustrated in Fig. 6. Threads within the block will update values in this array by using shared-memory atomics. When all threads of the block have finished, the private arrays are written out into the global density array with global atomics.

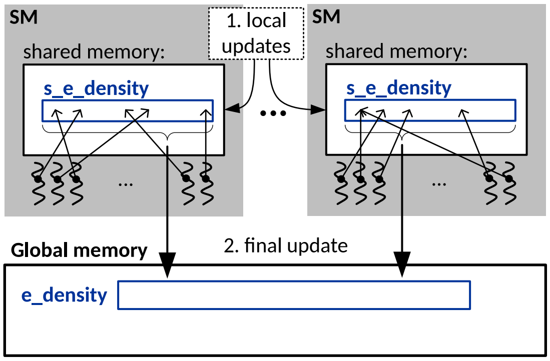

Fig. 6. Outline of the optimised density calculation using local density arrays for each block. Once each block completed its density update using shared atomics, the shared memory arrays are written out into the main density variable in global memory with global atomic operations.

```
// block-local density vector
extern __shared__ float s_e_density[];
for (int i = threadIdx.x; i < n; i += blockDim.x)
  s_e_density[i] = 0.0f;  // initialise to zero
__syncthreads(); // wait for all threads to complete
...
float xei = x_e[tid];
int p = (int)(xei/deltax);
// local update in shared memory density array
atomicAdd(&(s_e_density[p]), ((p+1)*deltax-xei)*c_de_factor);
atomicAdd(&(s_e_density[p+1]), (xei-p*deltax)*c_de_factor);
__syncthreads(); // wait for all threads to complete
int k = threadIdx.x;
while (k < n) {  // update global memory density array
  atomicAdd(&(e_density[k]), s_e_density[k]);
  k += blockDim.x;
}
```

While the code has become more complex, there are several benefits; updates are now performed in the much faster shared memory, and the number of global atomic operations is reduced from the number of particles to the number of grid cells. The final code shows increase in arithmetic intensities (3.98 and 4.22) and a significant improvement in performance (275.4 and 280 Gflop/s).

4.6. Poisson solver

Traditionally, the 1D Poisson equation is solved on CPUs using the well-known Thomas algorithm [50] that provides a solution in $O(n)$ steps, where n is the number of unknowns, i.e. the number of grid points. The Thomas algorithm is very efficient, requires little memory space but is inherently sequential. Consequently, in several GPU simulation implementations a hybrid execution scheme is used, i.e. the Poisson equation is solved on the CPU after the input data are copied back from the GPU memory, then once the solution is computed, the results are transferred back to the GPU memory for continuing the simulation loop.

In our implementation, we aimed to execute the solver on the GPU using a parallel algorithm to remove the data transfer overhead and make use of the parallelism the GPU offers. Two parallel solver versions were implemented and tested. The first is based on the Parallel Cyclic Reduction algorithm [51]. Our CUDA implementation is based on the GPU version introduced in [52]. The second approach that we used in our OpenCL implementation is based on the direct solution of the discrete Poisson equation. This approach requires more memory space and requires $O(n^2)$ steps for solution, but can be executed in parallel efficiently using matrix–vector operations.

Table 6
Effect of kernel performance optimisations on arithmetic intensity (AI), streaming multiprocessor (SM) and memory subsystem utilisation.

Kernel	AI		SM util. (%)		Memory util. (%)	
	Original	Optimised	Original	Optimised	Original	Optimised
electron_move	1.18	1.18	4.29	21.24	15.29	69.91
electron_collision	0.65	0.65	28.03	28.03	71.97	71.97
electron_density	2.69	3.98	1.01	13.45	5.25	15.87
ion_move	1.16	1.16	4.28	21.64	15.71	73.68
ion_collision	2.92	2.92	47.18	47.18	76.87	76.87
ion_density	3.98	4.22	1.22	13.69	6.28	14.30
Poisson_solver	0.05	0.05	0.41	0.41	6.1	6.1

Taking the Discrete Poisson equation (5), the solution is obtained by solving a set of linear equations which can be collectively written in the matrix form

$$\overline{\overline{C}}\phi_p = \frac{\rho_p \Delta x^2}{\varepsilon_0}, \tag{19}$$

where $\overline{\overline{C}}$ is the coefficient matrix of the linear system and ρ_p is the charge density at pth grid point ($p = 1, 2, \ldots, M-2$). The solution can be written as

$$\phi_p = \overline{\overline{C}}^{-1} \frac{\rho_p \Delta x^2}{\varepsilon_0}. \tag{20}$$

The boundary conditions at the edges of the computational domain (the first and the last grid point) are the externally imposed potentials of the electrodes. This means we only need to solve $M-2$ linear equations for $p = 1, 2, \ldots, M-2$.

The inverse matrix $\overline{\overline{C}}^{-1}$ is constant (and symmetric) throughout the simulation, and can be precomputed as

$$C_{sp} = \begin{cases} -\frac{(s+1)(M-p)}{M-1} & s \le p \\ C_{ps} & s > p \end{cases} \tag{21}$$

This allows us to solve Eq. (20) as a general matrix–vector multiplication operation, which has a highly efficient implementation for GPUs.

The execution time of the Parallel Cyclic Reduction Poisson solver kernel for the grid sizes 128, 256 and 512 (benchmark cases 1–4) on the V100 system are 8.84, 9.61, 11.15 and 11.13 µs, respectively. The OpenCL direct solver kernel times (WX9100) are 18.6, 19.4, 22.9 and 22.7 µs. The sequential CPU execution times using the Thomas algorithm are 2.7 (M = 128), 3.3 (M = 256) and 5.2 µs (M = 512) which are better than the GPU results but the overall time that includes the required additional device–host–device data transfer as well is in the range of 80–190 µs.

4.7. Kernel fusion

In the preceding parts we looked at the performance critical points and optimisation opportunities of the individual kernels of the PIC/MCC GPU simulation program. Table 6 summarises the effects of these performance improvements in terms of arithmetic intensity, streaming multiprocessor and memory utilisation levels.

The complete simulation program has three main parts: (i) initialisation, (ii) the simulation loop, and (iii) saving states and results to file system. The initialisation stage includes data input from files and copying data to GPU memory; the simulation loop executes GPU kernels for the specified number of simulation steps; the post-processing steps include copying results from GPU to host memory then saving the results to files.

As the length of the simulation and the number of particles are changing, the relative contribution (weight) of the kernels and the host–device data transfer operations to the overall execution time changes too. Table 7 lists these relative contribution values. The purpose of this analysis is to identify kernels or memory operations that might become potential performance bottlenecks.

Table 7
The relative contribution of individual kernels and data transfers to the overall simulation execution time. Each run is performed for 100 RF voltage cycles (V100 results).

Kernel	Relative kernel execution times (%)			
	Case 1	Case 2	Case 3	Case 4
electron_move	7.17	7.25	7.66	8.18
electron_collision	14.25	11.89	13.17	12.81
electron_density	9.02	15.58	19.26	20.20
ion_move	7.67	7.33	8.37	8.95
ion_collision	14.15	12.35	12.13	12.40
ion_density	9.21	16.35	19.98	20.94
Poisson_solver	22.31	18.55	12.45	11.80
host-to-device copy	5.57	2.46	0.94	0.41
device-to-host copy	10.58	8.17	5.05	4.30

The particle move and density kernels increase, while the Poisson solver and the memory copy operations decrease in weight as the number of particles and simulation steps increase (Case 1–4). The weight of the collision kernels remain approximately the same for the different benchmarks. Overall, the particle-related kernels (electron and ion move, collision and density) represent an increasing share – from 61.5 (Case 1) to 83.5% (Case 4) – of the simulation time, indicating that further optimisations might have significant effect in reducing the simulation time.

The main performance limiting factor in the particle kernels is the low arithmetic intensity. The number of arithmetic instructions cannot be increased in these kernels but if multiple kernels are fused into a single kernel, a number of load/store instructions can be reduced. For instance, the electron position is required in the e_move, e_collision and e_density kernels. In the fused kernel, we need to load the position only once, reducing memory transfer by a factor of 3. Based on this idea, we created two fused kernels, one for the electrons (`electron_kernel`), one for the ions (`ion_kernel`). Table 8 lists various performance metrics of the fused kernels obtained by executing benchmark case 4. The result of kernel fusion is further time reduction. For Case 4, the execution time of the fused kernels reduced to 82.0 (electron) and 77.64 (ion) µs from the original 117.2 and 114.1 µs produced by the separate electron and ion (move+collision+density) kernels. Measurements were made on a P100 card. Fig. 7 illustrates the effects of individual kernel optimisations and kernel fusion graphically on the roofline plot. The final, optimised fused kernels show large improvement in performance and move close to the memory-bound limit of the roofline performance curve. As a consequence, the production version of our CUDA and OpenCL PIC/MCC simulation codes are based on the fused kernel implementations.

5. Results and discussions

In this section, we first verify the correctness of our final, fused-kernel implementations by comparing them to the four

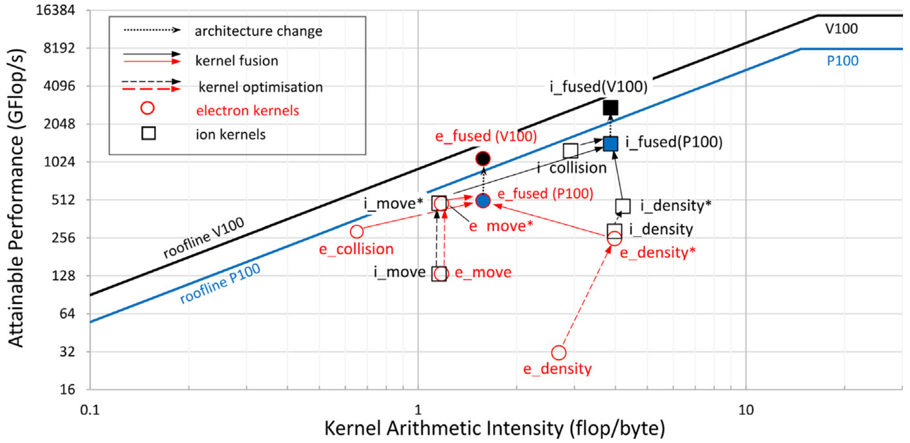

Fig. 7. Effects of the different optimisation steps shown on the roofline model. Red markers represent electron, while black ones represent ion kernel performance values. Arrows indicate how kernel arithmetic intensity and attained performance changed due to kernel level optimisations (dashed), kernel fusion (solid) and changing the target GPU architecture (dot-line). Asterisk marks kernel performance achieved with kernel-level optimisations.

Table 8
Performance metrics of the fused electron and ion kernels executing benchmark case 4.

Performance metric	Electron kernel	Ion kernel
Arithmetic intensity	1.58	3.87
kernel time P100 (μs)	82.0	77.64
kernel vs. total time (%)	45.93	43.48
performance P100 (Gflop/s)	207.9	488.0
performance V100 (Gflop/s)	650.3	1657.4

verified benchmark cases described in Section 2.1 (more details in [7]), then examine and compare the execution performance of the parallel CUDA and OpenCL versions on the selected GPU cards.

5.1. Verification of the GPU implementations

Fig. 8 shows the computed He$^+$ ion densities for the four benchmark cases obtained by both the CUDA and OpenCL implementations of the simulation. Comparing these data to the results from the original work [7] shows that both GPU implementations differ not more than 2% from each other and the original data. These deviations can originate from small differences in the precision of floating point arithmetic operations on the different architectures and the fact that the CPU implementation utilises only double precision (FP64) number representation, while the GPU codes use a mixture of single (FP32) and double precision numbers. Further, the charged particle density in the simulation is a statistically fluctuating quantity, which is determined by charge production in the plasma volume and losses at the electrodes, which processes are separated in space and often are characterised by a highly nonlinear time evolution.

5.2. Attained performance

As a comparison baseline, we list in Table 9 the execution times of the four sequential benchmark cases measured on two representative CPUs. The runtime values are given for 100 RF voltage cycles after steady-state is reached. Based on the number of simulation steps (Table 1) required for reaching steady-state, the overall sequential simulation time of these benchmarks are 3.8 min, 1.9 and 9.9 h, and 3.4 days, for Case 1, 2, 3 and 4, respectively. For Case 2 we also added two extra sub-cases using 1 and 10 million particles per species which are expected to provide more precise results (see our discussion on the undesired effect of the number of superparticles used in the simulations, in Section 2).

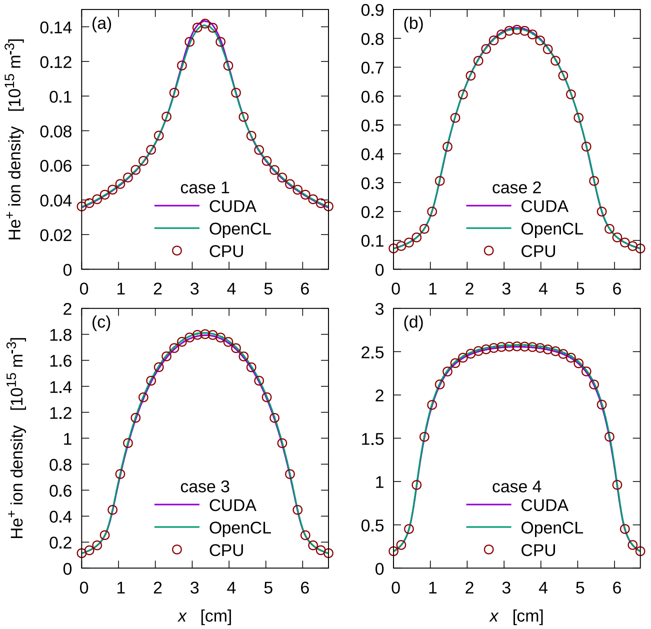

Fig. 8. He$^+$ ion density distributions of the four benchmark cases obtained by both the CUDA and OpenCL implementations on GPUs (lines) together with the original data as in [7] from the sequential CPU code (open circles). The data represent averages accumulated during the simulation of 10 000 RF cycles.

Table 9
Execution times (seconds) of the sequential benchmark cases 1 to 4 on Intel CPUs as indicated. Intel C compiler version 19.0 with compiler option "-fast" was used. Each run is for 100 RF voltage cycles.

Case	Particle counts	Time (seconds)	
	(electrons/ions)	Xeon Gold 6132	i7-6850K
1	12.3k/19.3k	18.7	19.5
2	57k/60.2k	131.9	133.5
3	138.7k/142.3k	700.4	706.6
4	161.6k/164.5k	1902	1897
2 (1M)*	987k/1040k	2945	2754
2 (10M)*	9.9M/10.4M	29 069	28 409

5.2.1. Strong scaling

First we examine the performance of the GPU implementation for the same problem sizes as in the sequential cases to determine

Table 10
GPU execution times (seconds) of the benchmark cases on various GPUs. Each run is for 100 RF cycles. The last row lists the highest achieved speedup values.

Card	Case 1	Case 2	Case 3	Case 4
NVIDIA GTX 1080 Ti	1.87	6.49	21.0	46.8
NVIDIA GTX 1660 Super	1.83	6.39	21.0	46.8
NVIDIA RTX 2080 Ti	1.83	4.30	13.5	30.6
NVIDIA Tesla P100	2.30	7.95	29.0	63.8
NVIDIA Tesla V100	1.99	4.36	13.7	28.5
AMD Radeon Pro WX9100	3.12	10.4	41.9	88.1
AMD Radeon VII	1.83	6.16	25.9	50.8
Speedup range	6.0–10.2	12.7–30.6	16.7–51.9	29.8–66.7

Table 11
Attained performance (in Gflop/s) of the fused ion computation kernel. In brackets is the same value expressed as percentage of peak performance (%).

Card	Case 1	Case 2	Case 3	Case 4
Tesla P100	516.6 (6.4%)	450.3 (5.6%)	492.5 (6.1%)	501.5 (6.2%)
Tesla V100	815.5 (5.5%)	1406 (9.4%)	1728 (11.6%)	1791 (12%)

to what extent the GPU versions reduce the overall simulation execution time. Table 10 lists the parallel execution times obtained with the CUDA and OpenCL implementations on our selected GPU cards. Results are given for 100 RF voltage cycles. The highest achieved speedup values based on the Xeon 6132 CPU execution times of Table 9 for cases 1 to 4 are 10.2, 30.6, 51.9, and 66.7, respectively. The speedup for Case 4, for instance, enables one to complete a simulation that originally ran for 1 week in about 2.5 h.

The results indicate that the execution times achieved on the different cards are in the same order of magnitude, but the CUDA implementations on the NVIDIA cards perform slightly better than the OpenCL versions on the AMD cards. Two cards, the Tesla P100 and the Radeon Pro WX9100, showed significantly weaker performance than the rest.

The most likely reason for the poorer OpenCL performance is the lack of floating point atomics. As a workaround, a loop-based implementation using integer atomic compare-exchange operations had to be used that incur performance penalties, especially at higher particle numbers.

Finally, in Table 11, we show the highest kernel performance achieved in the benchmark simulation cases along with the percentage of peak performance for two NVIDIA cards, the best performer V100 and the worst performer P100. The values are not particularly high but we must notice that for an Arithmetic Intensity value 3.87, one cannot expect to achieve more than 15%–25% of the peak performance.

5.2.2. Weak scaling

The trend of the increasing speedup and kernel performance values call for examining the performance of the GPU implementations for increasing particle counts. Under weak scaling, we are mainly interested in the size of a system we can simulate in the same time as in the sequential simulation.

We have measured the execution time of the benchmarks for 1 and 10 million particles per species (electrons and ions). Tables 12 and 13 list the GPU execution time results. Compared to the sequential CPU simulation times (Table 9, Xeon CPU), one can see that for 4 GPU cards for Case 1 and all cards for Cases 2–4 complete the 1M particle simulation faster than the sequential version for the original particle numbers. The highest speedup values for Cases 1–4 are 1.9, 6.1, 16.5 and 23.4. In the case of 10M particles, two cards finish faster than the sequential one for Case 3, and four cards for Case 4. The maximum speedup values for these cases are 2.2 and 3.2, respectively. The results show that we can execute more accurate simulations with increased particle counts and in many cases the results are still delivered faster than the sequential versions. Table 14 shows that this is due to the increase in relative kernel performance. The highest value, 2.735 Tflop/s is nearly 80% of the peak attainable performance given by the roofline model.

Table 12
GPU execution times (seconds) of benchmark cases 1–4 for 1 M particles/species. Each run is for 100 RF voltage cycles.

Card	Case 1	Case 2	Case 3	Case 4
NVIDIA GTX 1080 Ti	14.72	49.51	99.78	193.4
NVIDIA GTX 1660 Super	20.66	47.17	99.79	194.3
NVIDIA RTX 2080 Ti	11.14	24.07	49.90	97.77
NVIDIA Tesla P100	18.52	71.33	144.3	278.7
NVIDIA Tesla V100	9.66	21.52	42.41	81.13
AMD Radeon Pro WX9100	43.49	99.49	221.31	430.62
AMD Radeon VII	22.27	48.03	116.32	223.36

Table 13
GPU execution times (seconds) of benchmark cases 1–4 for 10 M particles/species. Each run is for 100 RF voltage cycles.

Card	Case 1	Case 2	Case 3	Case 4
NVIDIA GTX 1080 Ti	133.1	454.1	925.5	1733
NVIDIA GTX 1660 Super	195.6	442.6	931.5	1753
NVIDIA RTX 2080 Ti	93.66	201.3	408.3	758.7
NVIDIA Tesla P100	163.2	662.8	1335	2511
NVIDIA Tesla V100	81.86	167.7	323.5	594.5
AMD Radeon Pro WX9100	413.27	966.06	2097.34	4258.53
AMD Radeon VII	208.46	460.45	1080.60	2207.14

Table 14
Attained performance (in Gflop/s) of electron and ion kernels executing for 1 M and 10 M particles/species. In brackets the same value as fraction of peak performance (%).

Kernel	Case 1	Case 2	Case 3	Case 4
Electrons 1M	1011 (6.8%)	997 (6.7%)	1011 (6.8%)	1020 (6.8%)
Ions 1M	2526 (16.9%)	2418 (16.2%)	2410 (16.2%)	2408 (16.2%)
Electrons 10M	1147 (7.7%)	1105 (7.4%)	1133 (7.6%)	1174 (7.9%)
ions 10M	2735 (18.4%)	2611 (17.5%)	2624 (17.6%)	2626 (17.6%)

5.3. Performance vs. workload size

In order to examine the effect of increasing particle numbers on the performance in more detail, and to compute speedup values relative to the sequential version, we have selected one benchmark – Case 2 – for a particle number dependent analysis. The reason for using this benchmark is that this represents a physically relevant discharge system with a well developed quasi-neutral plasma bulk (in contrast to Case 1) and the runtimes for large particle numbers are still manageable on CPUs (in contrast to Cases 3 and 4). Particle number for each species were changed from 10k to 10M both in the sequential CPU and the parallel GPU versions. The results are summarised in Fig. 9. The top panel contains a plot of the CPU execution times for three different CPU models. The execution times are normalised by the number of executed RF cycles (100 cycles) and the total particle count ($N = N_e + N_i$). While there are obvious speed differences among the processors, the most notable phenomenon is the step-like increase of the normalised execution times. Up to about 10^5 particles the per-particle execution time is practically constant but after that the values transit rapidly to another, nearly constant plateau with normalised execution times about 30% higher. We suspect that this is due to caching effects. Until all particles fit in the L3 cache, the performance is ideal. Once particle vectors become too large to be cached entirely, cache invalidations start to happen. To support this hypothesis, in Fig. 9(a) the threshold particle number values at which the particle coordinate buffers

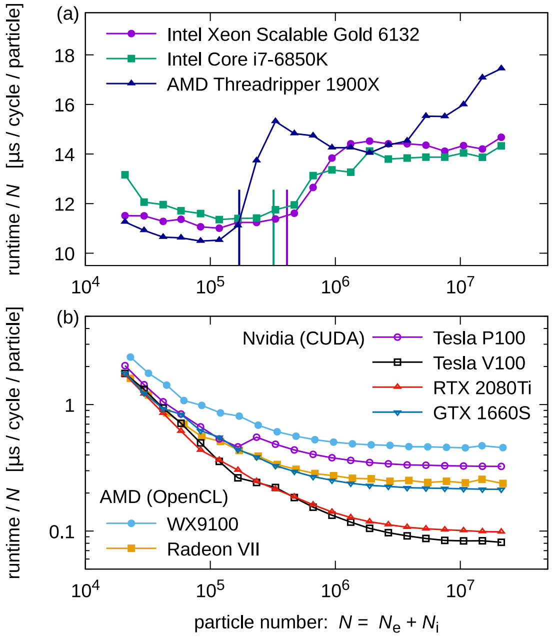

Fig. 9. CPU and GPU execution times in function of particle count. The measured runtimes are normalised by the number of the executed RF cycles (100 in every case) and the total particle count. The particle numbers are controlled by setting the superparticle weight factor. Panel (a) shows results of the original sequential implementation as introduced in [7] on 3 selected CPUs: Intel Xeon Scalable Gold 6132 with 19 712 kB L3 cache, Intel Core i7-6850K with 15 360 kB L3 cache, and AMD Ryzen Threadripper 1900X with 8192 kB of the same. The vertical lines indicate the particle number thresholds at which the particle buffers reach the size of the L3 cache for each CPU. Panel (b) shown the normalised runtimes of the CUDA and OpenCL implementations on different GPUs.

fill the entire L3 cache for each CPU is indicated with vertical bars. The position of the step-like runtime increase follows the order and relative distance of these threshold values.

The bottom panel of Fig. 9 shows the execution times obtained on the GPU cards. Here a trend opposite to the CPUs can be detected. For small particle numbers the per-particle execution time is high but it decreases as the particle count increases. After reaching a card-specific particle count threshold, the curves flatten and the per-particle time becomes constant. When this constant time is reached, the GPU chip is fully utilised, i.e. there are sufficient number of threads to keep the GPU busy all the time. We can observe that for most cards this threshold is around $N = 10^6$, while for the V100 and RTX 2080Ti cards it is closer to $N = 10^7$ particles. These results explain why we achieved increasing absolute performance as we increased the particle numbers.

From the CPU and GPU execution times we compute the speedup as function of particle count as $S_p(N) = T_{CPU}(N)/T_{GPU}(N)$. The result (the strong scaling speedup as function of particle count) is plotted in Fig. 10. Since the speedup depends on the speed of the CPU used, the absolute value of the speedup might be misleading (slower CPUs produce higher speedups), therefore we are focusing on the shape of the speedup function instead. For the sake of clarity, in Fig. 10 only one speedup curve is shown per GPU, the one calculated with our fastest server class CPU (Xeon Gold 6132), establishing a lower bound on the achieved speedup.

The speedup curves show several distinct trends. At low particle counts where the CPU works efficiently mainly from cache and the GPU is severely underutilised, the speedup values are modest. As the number of particles increases (10^5–10^6), the CPU becomes less efficient due to cache misses while the utilisation of the GPU increases, resulting in increasing speedup values. Finally, for large particle numbers, above 1 M particles, both CPU and GPU work at constant per-particle calculation speed resulting in a constant speedup. The actual speedup value depends on the CPU used as reference, as well as on the actual GPU card model, therefore we recommend the *relative* attained performance (percentage of the peak performance) as a parameter to compare parallel implementations.

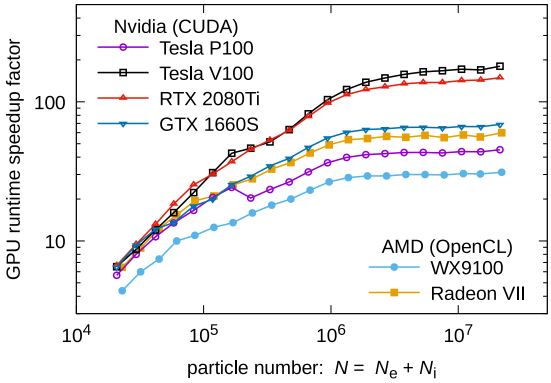

Fig. 10. GPU speedup ratios with respect to the performance of the sequential CPU implementation on the fastest CPU (Intel Xeon Scalable Gold 6132) as shown in Fig. 9(a).

The execution times given in Table 10 are based on running 100 RF voltage cycles only. The overall time for a simulation to reach steady state requires more RF cycles, typically between 1000 and 20 000, where (number of RF cycles $= N_s/\Delta t$, see Table 1). For benchmark cases 1–3 with parameters set in Table 1, our GPU implementations reach steady state typically within 1-35 min (time varies with benchmark cases and GPU cards used). For Case 4, execution times reach 1-2 h. The length of these simulation runs allow for one-shot executions where the entire simulation can be finished in one simulation run and state saving is sufficient only at the end of the program, the cost of which is negligible compared to the GPU execution time.

For larger simulations, however, having e.g. 1 or 10 million superparticles, the overall GPU execution time can increase up to 3 (Case 3) and 25 (Case 4) hours. In these cases, application level checkpoint/restart mechanisms with regular state saving (e.g. once every 30 or 60 min) should be employed to provide fault-resilient execution. Fortunately, GPU kernels execute concurrently with the host program, consequently, state saving can be completely overlapped in time with the execution of simulation kernels on the GPU, reducing checkpointing overhead effectively to zero.

Our results demonstrate that a single GPU card can execute 1D PIC/MCC simulations very efficiently. With sufficiently high number of particles, one can reach close to the attainable peak performance. The per-particle computation time also becomes closer to the theoretical minimum as the particle count increases. Although the focus of this paper was on 1D plasma simulation, our findings are also applicable to higher dimensions. Assuming a 1000 × 1000 grid and 200 particles per grid cell, a 2D simulation can fit entirely into the memory of high end cards, such as the AMD Radeon VII, NVIDIA Titan RTX, RTX3090, V100 or A100, enabling performing 2D simulation on (i) single cards and (ii) without any host-GPU data transfer penalty. For cards with smaller memory, unified memory can be used that stores particle data in host memory and transfers it to the device only when needed. In this case, some performance penalty is expected. We do not consider 3D PIC/MCC practical on single-card systems.

If the performance of a single card is not sufficient, multi-GPU systems can be used. The fact that the per-particle GPU execution time for 1D simulations is nearly constant above 1M particles means that large 1D simulations are expected to fully utilise the GPUs in multi-GPU configurations. Even with the added overhead of particle sorting due to domain-decomposition, with over 10M particles, one can expect further speedups when using 2, 4 or even 8 GPU cards per simulation. Studying the scalability of our implementation in multi-GPU configurations is an important future direction of our research.

We have shown that an architecture and performance-oriented view during the design and implementation of the PIC/MCC simulation is essential for achieving high performance. This does not mean our implementation is perfect. We believe there are still further optimisation opportunities and some parts of our code can be improved. Nevertheless, we demonstrated both with CUDA and OpenCL implementations that it is possible to reach sustained performance in the teraflops range and reduce execution time by two orders of magnitude. New architectural developments are also in favour of GPU-based PIC/MCC simulations. As the number of cores, computational performance and memory bandwidth increase in future GPU generations, the large number of threads used in our implementations accommodates for future growth, scaling and automatic improvement in simulation performance without any further code modification.

6. Conclusions

In this article, we described an efficient massively parallel GPU implementation for one-dimensional electrostatic PIC/MCC plasma simulations. Four plasma benchmark cases were used to verify the accuracy, correctness and performance of our implementation.

We described a particle-per-thread execution strategy with several performance optimisation steps to improve individual kernel performances. Using the roofline performance model, we could identify the performance boundaries of our program and operations that limit performance. Using within-kernel optimisations and kernel fusion, the final version achieved over 2.6 Tflop/s sustained performance and speedup values from few 10x to 200x depending on the number of particles. These results can have an important impact on advancing the science and practice of 1D kinetic electrostatic plasma simulations.

We implemented our simulations using CUDA 10.2 and OpenCL 2.0 and tested them on a representative set of NVIDIA and AMD GPU cards. In general, we found that OpenCL and CUDA implementations performed very similarly at kernel level execution. Also, the top-end AMD and NVIDIA cards also showed very similar performance. We have found some differences in the development environment and software support, however, with CUDA being more mature and better supported with libraries and development tools (IDEs, debuggers, profilers). We have also compared NVIDIA accelerator cards (V100 and P100) to consumer cards (GTX and RTX series) and found that consumer cards perform comparably or in some cases (P100) better than the significantly more expensive high-end server accelerator cards, which may pave the way for creating high performance simulation infrastructures at a moderate cost.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

JD and SM acknowledge support from the Slovak Research and Development Agency, project Nr. APVV-19-0386 and JD by Comenius University project Nr.: UK/180/2020. ZD, AD, and PH gratefully acknowledge financial support from the Hungarian Office for Research, Development and Innovation under NKFIH grants K-134462 and FK-128924. Support by the János Bolyai Research Fellowship of the Hungarian Academy of Sciences (AD) is acknowledged. ZJ was supported by the TKP2020-IKA-07 project financed under the 2020-4.1.1-TKP2020 Thematic Excellence Programme by the National Research, Development and Innovation Fund of Hungary. NVIDIA Corp. donated the Titan Xp GPU card that was used for program development.

References

[1] M.W. Evans, F.H. Harlow, The Particle-in-Cell method for Hydrodynamic Calculations, Technical Report LA-2139, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, 1957, URL https://apps.dtic.mil/dtic/tr/fulltext/u2/a384618.pdf.

[2] C. Birdsall, A. Langdon, Plasma Physics via Computer Simulation, CRC Press, Boca Raton, 1991, http://dx.doi.org/10.1201/9781315275048.

[3] C.K. Birdsall, L. Fellow, IEEE Trans. Plasma Sci. 19 (2) (1991) 65–85, http://dx.doi.org/10.1109/27.106800.

[4] M.A. Lieberman, A.J. Lichtenberg, Principles of Plasma Discharges and Materials Processing, John Wiley & Sons, 2005.

[5] P. Chabert, N. Braithwaite, Physics of Radio-Frequency Plasmas, Cambridge University Press, 2011, pp. i–iv.

[6] T. Makabe, Z.L. Petrovic, Plasma Electronics: Applications in Microelectronic Device Fabrication, Vol. 26, CRC Press, 2014.

[7] M.M. Turner, A. Derzsi, Z. Donkó, D. Eremin, S.J. Kelly, T. Lafleur, T. Mussenbrock, Phys. Plasmas 20 (1) (2013) 1–11, http://dx.doi.org/10.1063/1.4775084, arXiv:arXiv:1211.5246v2.

[8] P.C. Liewer, V.K. Decyk, J. Comput. Phys. 85 (2) (1989) 302–322, http://dx.doi.org/10.1016/0021-9991(89)90153-8.

[9] V.K. Decyk, Comput. Phys. Comm. 87 (1–2) (1995) 87–94, http://dx.doi.org/10.1016/0010-4655(94)00169-3.

[10] K.Z. Ibrahim, K. Madduri, S. Williams, B. Wang, S. Ethier, L. Oliker, Int. J. High Perform. Comput. Appl. 27 (4) (2013) 454–473, http://dx.doi.org/10.1177/1094342013492446.

[11] X. Sáez, A. Soba, E. Sánchez, M. Mantsinen, S. Mateo, J.M. Cela, F. Castejón, J. Phys. Conf. Ser. 640 (1) (2015) 12064, http://dx.doi.org/10.1088/1742-6596/640/1/012064.

[12] I. Surmin, S. Bastrakov, E. Efimenko, A. Gonoskov, A. Korzhimanov, I. Meyerov, Comput. Phys. Comm. 202 (2016) 204–210, http://dx.doi.org/10.1016/j.cpc.2016.02.004.

[13] A. Beck, J. Derouillat, M. Lobet, A. Farjallah, F. Massimo, I. Zemzemi, F. Perez, T. Vinci, M. Grech, Comput. Phys. Comm. (2019) http://dx.doi.org/10.1016/J.CPC.2019.05.001.

[14] W. Gropp, E.L. Lusk, A. Skjellum, Using MPI : Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, Mass, 1994, p. 307.

[15] L. Dagum, R. Menon, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.

[16] V.K. Decyk, C.D. Norton, Comput. Phys. Comm. 164 (1–3) (2004) 80–85, http://dx.doi.org/10.1016/j.cpc.2004.06.011.

[17] V.K. Decyk, T.V. Singh, Comput. Phys. Comm. 182 (3) (2011) 641–648, http://dx.doi.org/10.1016/j.cpc.2010.11.009.

[18] V.K. Decyk, T.V. Singh, Comput. Phys. Comm. 185 (3) (2014) 708–719, http://dx.doi.org/10.1016/J.CPC.2013.10.013.

[19] G. Chen, L. Chacón, D. Barnes, J. Comput. Phys. 231 (16) (2012) 5374–5388, http://dx.doi.org/10.1016/j.jcp.2012.04.040.

[20] X. Kong, M.C. Huang, C. Ren, V.K. Decyk, J. Comput. Phys. 230 (4) (2011) 1676–1685.

[21] F. Hariri, T. Tran, A. Jocksch, E. Lanti, J. Progsch, P. Messmer, S. Brunner, C. Gheller, L. Villard, Comput. Phys. Comm. (2016) http://dx.doi.org/10.1016/j.cpc.2016.05.008.

[22] G. Stantchev, W. Dorland, N. Gumerov, J. Parallel Distrib. Comput. 68 (10) (2008) 1339–1349, http://dx.doi.org/10.1016/j.jpdc.2008.05.009.

[23] D. Tskhakaya, R. Schneider, J. Comput. Phys. 225 (1) (2007) 829–839, http://dx.doi.org/10.1016/j.jcp.2007.01.002.

[24] A. Sun, M.M. Becker, D. Loffhagen, Comput. Phys. Comm. 206 (2016) 35–44, http://dx.doi.org/10.1016/j.cpc.2016.05.003.

[25] A. Fierro, J. Dickens, A. Neuber, APS (2013) ET2–005.

[26] A. Fierro, J. Dickens, A. Neuber, Phys. Plasmas 21 (12) (2014) 123504.
[27] H. Shah, S. Kamaria, R. Markandeya, M. Shah, B. Chaudhury, 2017 IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, pp. 378–387.
[28] I. Sohn, J. Kim, J. Bae, J. Lee, IEEE Trans. Plasma Sci. 44 (9) (2016) 1823–1833, http://dx.doi.org/10.1109/TPS.2016.2593491.
[29] J. Claustre, B. Chaudhury, G. Fubiani, M. Paulin, J. Boeuf, IEEE Trans. Plasma Sci. 41 (2) (2013) 391–399.
[30] M.Y. Hur, J.S. Kim, I.C. Song, J.P. Verboncoeur, H.J. Lee, Plasma Res. Express 1 (1) (2019) 015016.
[31] P. Mertmann, D. Eremin, T. Mussenbrock, R.P. Brinkmann, P. Awakowicz, Comput. Phys. Comm. 182 (10) (2011) 2161–2167, http://dx.doi.org/10.1016/j.cpc.2011.05.012, arXiv:1104.3998.
[32] N. Hanzlikova, Particle-in-Cell Simulations of Highly Collisional Plasmas on the GPU in 1 and 2 Dimensions (Ph.D. thesis), Dublin City University, Dublin City University, Dublin, Ireland, 2015.
[33] J.P. Verboncoeur, Plasma Phys. Control. Fusion 47 (5A) (2005) A231–A260, http://dx.doi.org/10.1088/0741-3335/47/5a/017.
[34] Z. Donkó, Plasma Sources. Sci. Technol. 20 (2) (2011) 024001, http://dx.doi.org/10.1088/0963-0252/20/2/024001.
[35] M.M. Turner, Phys. Plasmas 13 (3) (2006) 033506, http://dx.doi.org/10.1063/1.2169752, arXiv:https://doi.org/10.1063/1.2169752.
[36] E. Erden, I. Rafatov, Contrib. Plasma Phys. 54 (7) (2014) 626–634, http://dx.doi.org/10.1002/ctpp.201300047.
[37] Cross sections extracted from PROGRAM MAGBOLTZ, VERSION 7.1, Biagi-v7.1 database, 2004, www.lxcat.net (retrieved on February 20, 2017).
[38] A.V. Phelps, J. Appl. Phys. 76 (2) (1994) 747–753, http://dx.doi.org/10.1063/1.357820.
[39] Compilation of Atomic and Molecular Data 2005; http://jila.colorado.edu/{~}avp/.
[40] S. Chandrasekaran, G. Juckeland, OpenACC for Programmers: Concepts and Strategies, Addison-Wesley Professional, 2017.
[41] B. Chapman, G. Jost, R. Van Der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, Vol. 10, MIT press, 2008.
[42] A. Munshi, 2009 IEEE Hot Chips 21 Symposium (HCS), IEEE, 2009, pp. 1–314.
[43] A. Munshi, B. Gaster, T.G. Mattson, D. Ginsburg, OpenCL Programming Guide, Pearson Education, 2011.
[44] J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming, John Wiley & Sons, 2014.
[45] S. Cook, CUDA Programming: a Developer's Guide to Parallel Computing With GPUs, Newnes, 2012.
[46] S. Williams, A. Waterman, D. Patterson, Commun. ACM 52 (4) (2009) 65–76, http://dx.doi.org/10.1145/1498765.1498785.
[47] A. Lopes, F. Pratas, L. Sousa, A. Ilic, 2017 IEEE Int. Symp. Perform. Anal. Syst. Softw., IEEE, 2017, pp. 259–268, http://dx.doi.org/10.1109/ISPASS.2017.7975297.
[48] P. Hartmann, L. Wang, K. Nösges, B. Berger, S. Wilczek, R.P. Brinkmann, T. Mussenbrock, Z. Juhasz, Z. Donkó, A. Derzsi, E. Lee, J. Schulze, Plasma Sources. Sci. Technol. 29 (7) (2020) 075014, http://dx.doi.org/10.1088/1361-6595/ab9374.
[49] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes 3rd Edition: The Art of Scientific Computing, third ed., Cambridge University Press, USA, 2007.
[50] L.H. Thomas, Elliptic Problems in Linear Difference Equations Over a Network, Vol. 1, Watson Sci. Comput. Lab. Rept., Columbia University, New York, 1949.
[51] R. Hockney, C. Jesshope, Parallel Computers: Architecture, Programming and Algorithms, Adam Hilger, Bristol, 1981.
[52] Y. Zhang, J. Cohen, J.D. Owens, ACM SIGPLAN Not. 45 (5) (2010) 127, http://dx.doi.org/10.1145/1837853.1693472.